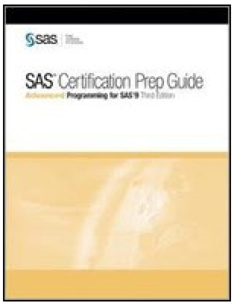

sas
SAS Certification Prep Guide
Advanced Programming for SAS 9 Third Edition

SAS Certification Prep Guide: Advanced Programming for SAS 9, Third Edition

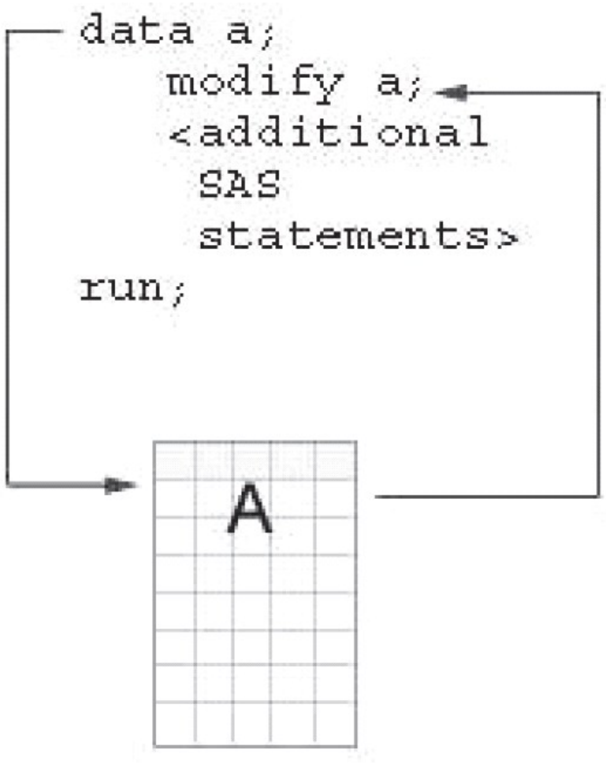
by SAS Institute

Chapter 18: Modifying SAS Data Sets and Tracking Changes

Overview

Introduction

There are times when you want to modify the observations in a SAS data set without replacing the data set. You can do this in a DATA step with the MODIFY statement. Using the MODIFY statement, you can replace, delete, or append observations in an existing data set without creating an additional copy of the data. In this chapter, you learn to modify all the observations in a data set, matching observations using a BY statement, and observations located using an index.



```
data a;
   modify a;
   <additional
     SAS
     statements>
run;
```

When you modify data, it's often essential to safeguard your data and track the changes that are made. In this chapter you learn how to create integrity constraints to protect your data. You will also learn to different methods of tracking changes — audit trails and generation data sets. You use audit trails to track changes that are made to a data set in place, and you use generation data sets to track changes that are made when the data set is rebuilt.

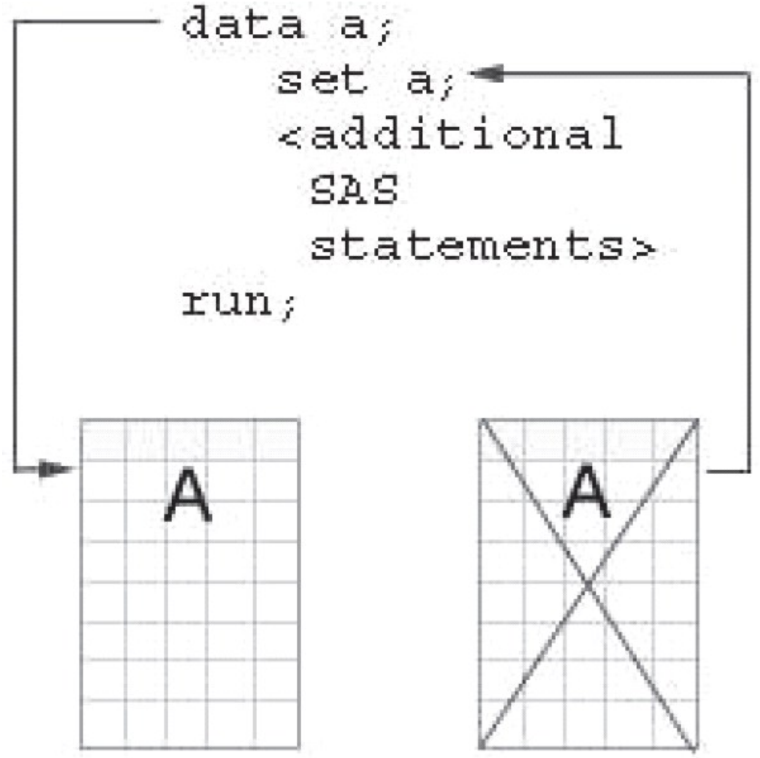
Objectives

In this chapter, you learn to

- use the MODIFY statement to update all observations in a SAS data set
- use a transaction data set to make modifications to a SAS data set
- use an index to locate observations to modify in a SAS data set
- place integrity constraints on variables in a SAS data set
- initiate and manage an audit trail file
- create and process generation data sets.

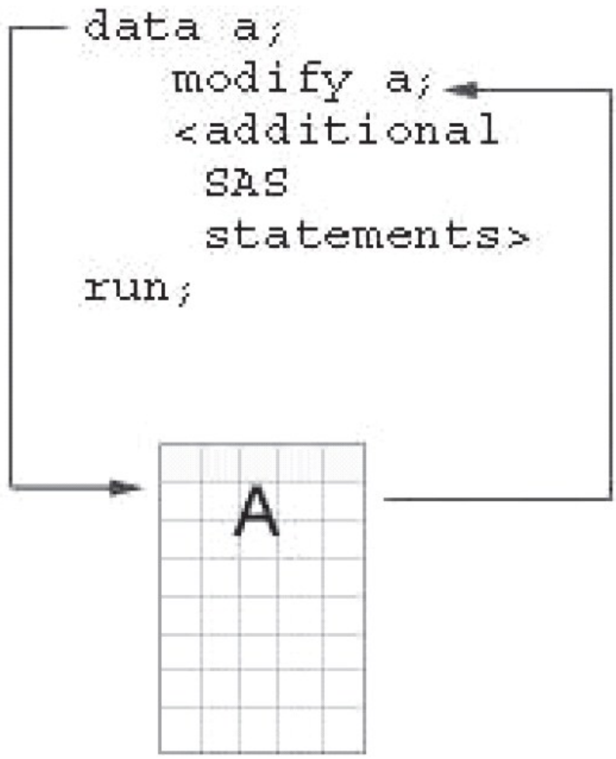
Using the MODIFY Statement

When you submit a DATA step to create a SAS data set that is also named in a *MERGE, UPDATE*, or *SET* statement, SAS creates a second copy of the input data set. Once execution is complete, SAS deletes the original copy of the data set. As a result, the original data set is replaced by the new data set. The new data set can contain a different set of variables than the original data set and the attributes of the variables in the new data set can be different from those of the original data set.



```
data a;
   set a;
   <additional
    SAS
    statements>
run;
```

In contrast, when you submit a DATA step to create a SAS data set that is also named in the *MODIFY statement*, SAS does not create a second copy of the data but instead updates the data set in place. Any variables can be added to the program data vector (PDV), but they are not written to the data set. Therefore, the set of variables in the data set does *not* change when the data is modified.



```
data a;
   modify a;
   <additional
    SAS
    statements>
run;
```

When you use the MODIFY statement, there is an implied REPLACE statement at the bottom of the DATA step instead of an OUTPUT statement. Using the MODIFY statement, you can update

- every observation in a data set
- observations using a transaction data set and a BY statement
- observations located using an index.

Caution If the system terminates abnormally while a DATA step that is using the MODIFY statement is processing, you can lose data and possibly damage your master data set. You can recover from the failure by

- restoring the master file from a backup and restarting the step, or
- keeping an audit trail file and using it to determine which master observations have been updated.

First we will consider using the MODIFY statement to modify all the observations in the data set.

Modifying All Observations in a SAS Data Set

Overview

When every observation in a SAS data set requires the same modification, you can use the MODIFY statement and specify the modification using an assignment statement.

General form, MODIFY statement with an assignment statement:

```
DATA SAS-data-set;
      MODIFY SAS-data-set;
      existing-variable = expression;
RUN;
```

where

SAS-data-set

is the name of the SAS data set that you want to modify.

existing-variable

is the name of the variable whose values you want to update.

expression

is a function or other expression that you want to apply to the variable.

Example

Suppose an airline has decided to give passengers more leg room. To do so the airline must decrease the number of seats in the business and economy classes. The SAS data set *Capacity* has the variables **`CapEcon`** and **`CapBusiness`** that hold values for the number of seats in the economy and business classes.

In the program below, the assignment statement for **`CapEcon`** reduces the number of seats in the economy class to 95% of the original number, and the assignment statement for **`CapBusiness`** reduces the number of seats in the business class to 90% of the original number. The INT function is used in both assignment statements to return the integer portion of the result.

Note If you choose to run this example, you must copy the data set *Capacity* from the *Sasuser* library to the *Work* library.

```
proc print data=capacity (obs=4);
run;

data capacity;
   modify capacity;
   CapEcon = int(CapEcon * .95);
   CapBusiness = int(CapBusiness * .90);
run;

proc print data=capacity (obs=4);
run;
```

The following output shows the data *before* the MODIFY statement.

Obs	FlightID	RouteID	Origin	Dest	Cap1st	CapBusiness	CapEcon
1	IA00100	0000001	RDU	LHR	14	30	163
2	IA00201	0000002	LHR	RDU	14	30	163
3	IA00300	0000003	RDU	FRA	14	30	163
4	IA00400	0000004	FRA	RDU	14	30	163

The following output shows the data *after* the MODIFY statement. You can see that the values in **`CapBusiness`** and **`CapEcon`** have been reduced.

Obs	FlightID	RouteID	Drigin	Dest	Cap1st	CapBusiness	CapEcon
1	IA00100	0000001	RDU	LHR	14	27	**154**
2	IA00201	0000002	LHR	RDU	14	27	**154**
3	IA00300	0000003	RDU	FRA	14	27	**154**
4	IA00400	0000004	FRA	RDU	14	27	**154**

Modifying Observations Using a Transaction Data Set

Overview

You can use a MODIFY statement to update all observations in a data set, but there are times when you only want to update selected observations. You can modify a *master* SAS data set with values in a *transaction* data set by using the MODIFY statement with a BY statement to apply updates by matching observations.

General form, MODIFY statement with a BY statement:

```
DATA SAS-data-set;
      MODIFY SAS-data-set transaction-data-set;
      BY key-variable;
RUN;
```

where

SAS-data-set

is the name of the SAS data set that you want to modify (also called the master data set).

transaction-data-set

is the name of the SAS data set in which the updated values are stored.

key-variable

is the name of the variable whose values will be matched in the master and transaction data sets.

Note In the MODIFY statement, you must list the master data set followed by the transaction data set.

The BY statement matches observations from the transaction data set with observations in the master data set. When the MODIFY statement reads an observation from the transaction data set, it uses dynamic WHERE processing (SAS internally generates a WHERE statement) to locate the matching observation in the master data set. The matching observation in the master data set can be replaced, deleted, or appended. By default, the observation is replaced.

Note Because the MODIFY statement uses WHERE processing to locate matching observations, neither data set requires sorting. However, having the master data set sorted or indexed and the transaction data set sorted reduces processing overhead, especially for large files.

Example

Suppose you have a master data set, *Capacity*, which has route numbers for an airline. Some of the route numbers have changed, and the changes are stored in a transaction data set, *Newrtnum*. The master data set is updated by matching values of the variable **`FlightID`**.

```
proc print data=capacity(obs=5);
run;

data capacity;
   modify capacity sasuser.newrtnum;
   by flightid;
run;

proc print data=capacity(obs=5);
run;
```

The following PROC PRINT output displays the first five rows of the data set *Capacity* before updates were applied.

Obs	FlightID	RouteID	Origin	Des!	Cap1st	CapBusiness	CapEcon
1	IA00100	0000001	RDU	LHR	14	30	163
2	IA00201	0000002	LHR	RDU	14	30	163
3	IA00300	0000003	RDU	FRA	14	30	163
4	IA00400	0000004	FRA	RDU	14	30	163
5	IA00500	0000005	RDU	JFK	16		251

As you can see in this PROC PRINT output, three values of **`FlightID`** in *Newrtnum* have matches in *Capacity*. For each matching observation, the values for **`RouteID`** are updated.

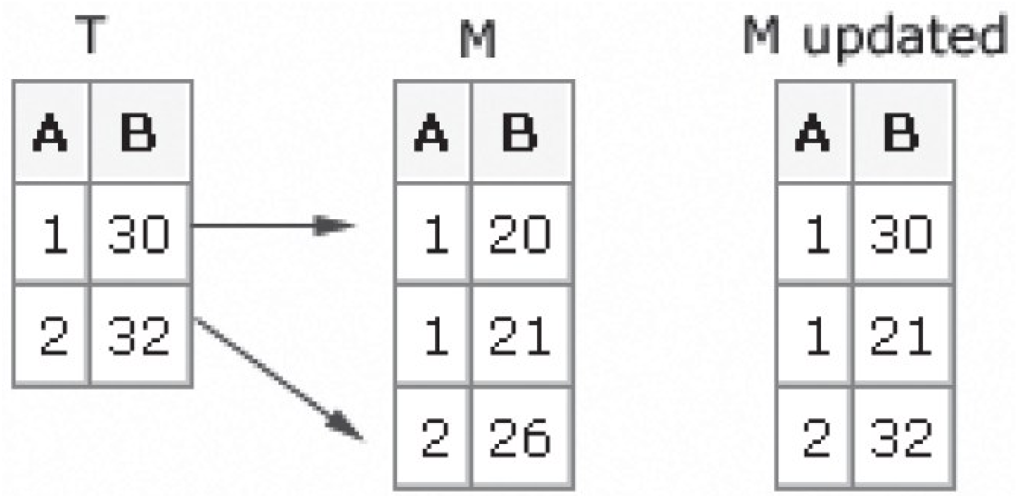
Obs	FlightID	RouteID	Origin	Dest	Cap1st	CapBusiness	CapEcon
1	IA00100	**`0000101`**	RDU	LHR	14	30	163
2	IA00201	0000002	LHR	RDU	14	30	163
3	IA00300	0000003	RDU	FRA	14	30	163
4	IA00400	**`0000400`**	FRA	RDU	14	30	163
5	IA00500	**`0000035`**	RDU	JFK	16		251

Handling Duplicate Values

When you use the MODIFY and BY statements to update a data set, WHERE processing starts at the top of the master data set and finds the first match and updates it. We will consider what happens if there are duplicate values in the master or transaction data sets. Suppose you have the following code to make updates to the master data set *M* using the transaction data set *T*:

```
data m;
    modify m t;
    by a;
run;
```

If duplicate values of the BY variable exist in the *master data set*, only the first observation in the group of duplicate values is updated because WHERE processing begins at the top of the data set and updates the first match.



T

A	B
1	30
2	32

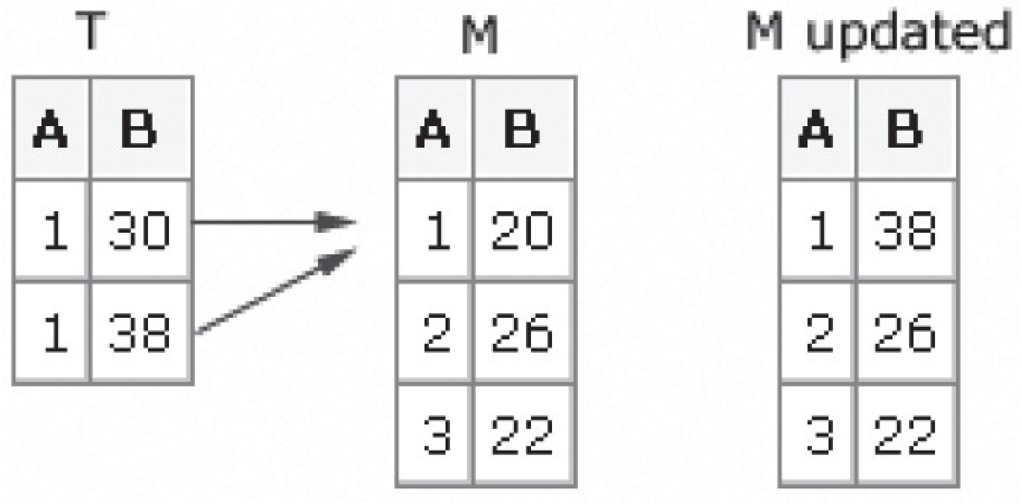
M

A	B
1	20
1	21
2	26

M updated

A	B
1	30
1	21
2	32

If duplicate values of the BY variable exist in the *transaction data set*, the duplicate values overwrite each other so that the last value in the group of duplicate transaction values is the result in the master data set.



T

A	B
1	30
1	38

M

A	B
1	20
2	26
3	22

M updated

A	B
1	38
2	26
3	22

You can avoid overwriting duplicate values by writing an accumulation statement so that all the observations in the transaction data set are added to the master observation.

Tip If duplicate values exist in both the *master and transaction data sets*, you can use PROC SQL to apply the duplicate values in the transaction data set to the duplicate values in the master data set in a one-to-one correspondence.

Handling Missing Values

If there are missing values in the transaction data set, SAS does not replace the data in the master data set with missing values unless they are special missing values.

Note A special missing value is a type of numeric missing value that enables you to represent different categories of missing data by using the letters A-Z or an underscore. You designate special missing values using the MISSING statement in the DATA step. For more information, see the SAS documentation.

You can specify how missing values in the transaction data set are handled by using the UPDATEMODE= option in the MODIFY statement.

General form, MODIFY statement with the UPDATEMODE= option:

```
MODIFY master-data-set transaction-data-set
             UPDATEMODE=MISSINGCHECK | NOMISSINGCHECK;
```

where

master-data-set

is the name of the SAS data set that you want to modify.

transaction-data-set

is the name of the SAS data set in which the updated values are stored.

MISSINGCHECK

prevents missing values in the transaction data set from replacing values in the master data set unless they are special missing values. MISSINGCHECK is the default.

NOMISSINGCHECK

allows missing values in the transaction data set to replace the values in the master data set. Special missing values in the transaction data set still replace values in the master data set.

Modifying Observations Located by an Index

Overview

You have learned that you can use a BY statement to access values you want to update in a master data set by matching. When you have an indexed data set, you can use the index to directly access the values you want to update. To do this, you use

- a MODIFY statement with the KEY= option to name an indexed variable to locate the observations for updating
- another data source (typically a SAS data set named on a SET statement or an external file read by an INPUT statement) to provide a like-named variable whose values are supplied to the index.

General form, MODIFY statement with the KEY= option:

`MODIFY` *`SAS-data-set`* **`KEY=`***`index-name`*`;`

where

SAS-data-set

is the master data set, or the data set that you want to update.

index-name

is the name of the simple or composite index that you are using to locate observations.

Updating with an index is different from updating using a BY statement. When you use the MODIFY statement with the KEY= option to name an index,

- you must *explicitly specify the update* that you want to occur. No automatic overlay of non-missing values in the transaction data set occurs as it does with the MODIFY/BY method.
- each observation in the transaction data set *must have a matching observation* in the master data set. If you have multiple observations in the transaction data set for one master observation, only the first observation in the transaction data set is applied. The other observations generate run time errors and terminate the DATA step (unless you use the UNIQUE option, which is discussed later in this chapter).

Example

Suppose that airline cargo weights for 1999 are stored in the master data set *Cargo99*, which has a composite index named *FlghtDte* on the variables **`FlightID`** and **`Date`**. Some of the data is incorrect and the data set needs to be updated. The correct cargo data is stored in the transaction data set *Newcgnum*.

In the program below, the KEY= option specifies the *FlghtDte* index. When a matching observation is found in *Cargo99*, three variables (**`CapCargo, Cargowgt`**, and **`CargoRev`**) are updated.

Note If you choose to run this example, you must copy the data set *Cargo99* from the *Sasuser* library to the *Work* library.

```
proc print data=cargo99(obs=5);
run;

data cargo99;
   set sasuser.newcgnum (rename =
       (capcargo = newCapCargo
       cargowgt = newCargoWgt
       cargorev = newCargoRev));
   modify cargo99 key=flghtdte;
   capcargo = newcapcargo;
   cargowgt = newcargowgt;
   cargorev = newcargorev;
run;

proc print data=cargo99(obs=5);
```

```
run;
```

The output below shows the first five observations of the SAS data set *Cargo99* before it as modified by *Newcgnum*.

Obs	FlightID	RouteID	Origin	Dest	CapCargo	Date	CargoWgt	CargoRev
1	IA00100	0000001	RDU	LHR	82400	01JAN1999	45600	$111,720.00
2	IA00100	0000001	RDU	LHR	82400	01AUG1999	44600	$109,270.00
3	IA00100	0000001	RDU	LHR	82400	20AUG1999	44600	$109,270.00
4	IA00100	0000001	RDU	LHR	82400	02SEP1999	47400	$116,130.00
5	IA00100	0000001	RDU	LHR	82400	29DEC1999	44200	$108,290.00

The output below shows the first five observations of the SAS data set *Cargo99* after it as modified by *Newcgnum*. Notice that the three variables in the first observation were updated by the values in *Newcgnum*.

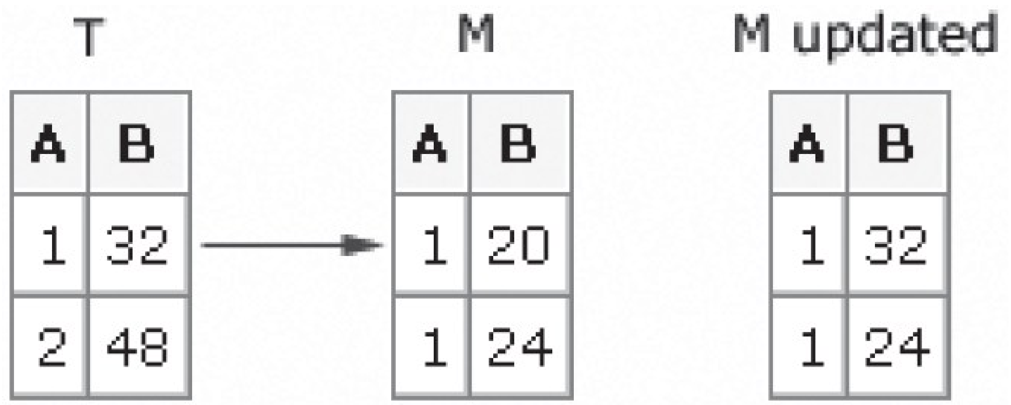
Obs	FlightID	RouteID	Origin	Dest	CapCargo	Date	CargoWgt	CargoRev
1	IA00100	0000001	RDU	LHR	**35055**	01JAN1999	.	**$121,879.90**
2	IA00100	0000001	RDU	LHR	82400	01AUG1999	44600	S109,270.00
3	IA00100	0000001	RDU	LHR	82400	20AUG1999	44600	$109,270.00
4	IA00100	0000001	RDU	LHR	82400	02SEP1999	47400	$116,130.00
5	IA00100	0000001	RDU	LHR	82400	29DEC1999	44200	$108,290 00

Handling Duplicate Values

When you use an index to locate values to update, duplicate values of the indexed variable in the transaction data set might cause problems. We consider what happens with various scenarios when you use the following code to update the master data set *M* with values from the transaction data set *T*. The index on the *M* data set is built on the variable **A**:

```
data m;
   set t (rename=(b=newb));
   modify m key=a;
   b=newb;
run;
```

If there are *duplications in the master data set*, only the first occurrence is updated.



T

A	B
1	32
2	48

M

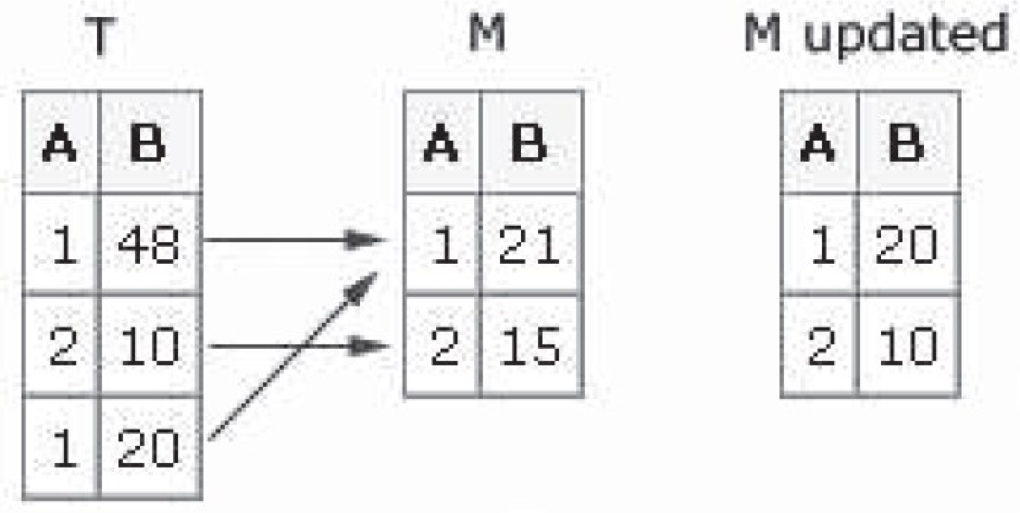
A	B
1	20
1	24

M updated

A	B
1	32
1	24

Tip If you want all duplicates in the master data set to be updated with the transaction value, use a DO loop to execute a SET statement with the KEY= option multiple times.

If there are *nonconsecutive duplications in the transaction data set*, SAS updates the first observation in the master data set. The last duplicate transaction value is the result in the master data set after the update.

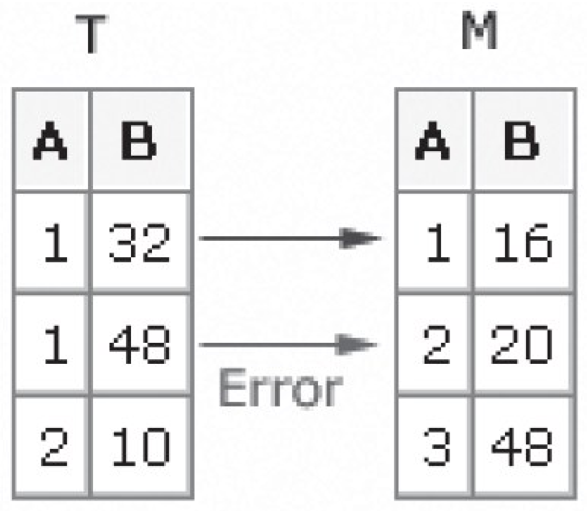


A	B
1	48
2	10
1	20

A	B
1	21
2	15

A	B
1	20
2	10

If there are *consecutive duplications in the transaction data set*, some of which do not have a match in the master data set, then SAS performs a one-to-one update until it finds a non-match. At that time, the DATA step terminates with an error.



A	B
1	32
1	48
2	10

A	B
1	16
2	20
3	48

Adding the *UNIQUE* option to the MODIFY statement allows you to avoid the error in the DATA step. The UNIQUE option causes the DATA step to return to the top of the index each time it looks for a match for the value from the transaction data set. The UNIQUE option can be used only with the KEY= option.

General form, MODIFY statement with the UNIQUE option:

MODIFY *SAS-data-set* **KEY**=*index-name* /**UNIQUE**;

where

SAS-data-set

positions the input pointer on a specified column.

index-name

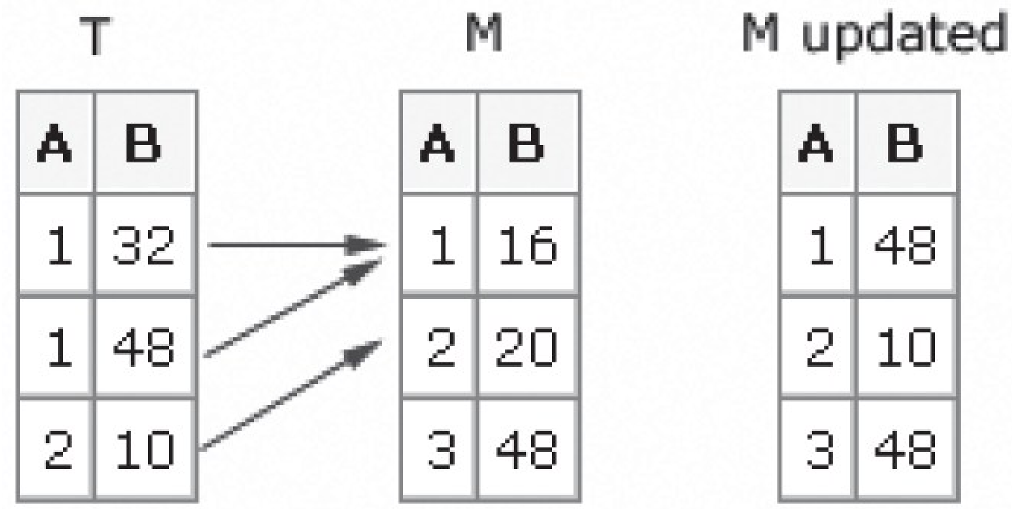
is the name of the variable that is being created.

You can specify the *UNIQUE* option to

- apply multiple transactions to one master observation

- identify that each observation in the master data set contains a unique value of the index variable.

When you use the UNIQUE option and there are consecutive duplications in the transaction data set, SAS updates the first observation in the master data set. This is similar to what happens when you have nonconsecutive duplications in the transaction data set. If the values in the transaction data set should be added to the value in the master data set, you can write a statement to accumulate the values from all the duplicates.



T

A	B
1	32
1	48
2	10

M

A	B
1	16
2	20
3	48

M updated

A	B
1	48
2	10
3	48

Controlling the Update Process

Overview

The way SAS writes observations to a SAS data set when the DATA step contains a MODIFY statement depends on whether certain other statements are present. If no other statements are present, SAS writes the current observation to its original place in the SAS data set. This action occurs by default through an implied REPLACE statement at the bottom of the DATA step.

However, you can override this default behavior by explicitly adding the *OUTPUT, REPLACE*, or *REMOVE* statement.

General form for OUTPUT, REPLACE, and REMOVE statements:

```
OUTPUT;
REPLACE;
REMOVE;
```

where

OUTPUT

specifies that the current observation be written to the *end* of the data set.

REPLACE

specifies that the current observation be reritten to the *same location* in the data set.

REMOVE

specifies that the current observation be *deleted* from the master data set.

Using OUTPUT, REPLACE, or REMOVE in a DATA step overrides the default replacement of observations. If you use any one of these statements in a DATA step, you must explicitly program each action that you want to take. You can use these three statements together as long as the sequence is logical.

Caution If you use an OUTPUT statement in conjunction with a REPLACE or REMOVE statement, be sure the OUTPUT statement is executed after any REPLACE or REMOVE statements to ensure the integrity of the index position.

Example

If the SAS data set *Transaction* has a variable named **`code`** having values of *yes, no*, and *new*, you can submit the following program to

- delete rows where the value of **`code`** is *no*
- update rows where the value of **`code`** is *yes*
- append rows where the value of **`code`** is *new*.

```
data master;
   set transaction;
   modify master key = id;
   a = b;
   if code = 'no' then remove;
   else if code = 'yes' then replace;
   else if code = 'new' then output;
run;
```

Note You cannot run this example because *Transaction* and *Master* are fictitious data sets.

Monitoring I/O Error Conditions

When you use the MODIFY statement with a BY statement or KEY= option to update a data set, error checking is important for several reasons. The most important reason is that these tools use nonsequential access methods, so there is no guarantee that an observation will be located that satisfies the request. Error checking enables you to perform updates or not, depending on the outcome of the I/O condition.

The *automatic variable* **`_IORC_`** (Input Output Return Code) is created when you use the MODIFY statement with the BY statement or KEY= option. The value of **`_IORC_`** is a numeric return code that indicates the status of the most recently executed I/O operation. Checking the value of this variable allows you to detect abnormal I/O conditions and direct execution in particular ways rather than having the application terminate abnormally.

Using _IORC_ with %SYSRC

Because the values of the **`_IORC_`** automatic variable are internal and subject to change, *%SYSRC*, an autocall macro, was created to enable you to test for *specific I/O conditions* while protecting your code from future changes in **`_IORC_`** values.

General form, _IORC_ with the %SYSRC autocall macro:

```
IF_IORC_=%SYSRC (mnemonic) THEN executable statement;
```

where

mnemonic

is a code for a specific I/O condition.

Note *%SYSRC* is in the autocall library. You must have the MACRO system option in effect to use this macro.

When you use *%SYSRC*, you can check the value of **`_IORC_`** by specifying one of the mnemonics listed in the table below.

Mnemonic	Meaning
_DSENMR	The observation in the transaction data set does not exist in the master data set (used only with the MODIFY and BY statements).

_DSEMTR	Multiple transaction data set observations do not exist on the master data set (used only with the MODIFY and BY statements).
_DSENOM	No matching observation (used with the KEY= option).
_SOK	The observation was located.

General form, _IORC_ with the %SYSRC autocall macro:

IF_IORC_=%SYSRC (*mnemonic*) **THEN**...

where

mnemonic

is a code for a specific I/O condition.

Example

Suppose you are using the MODIFY statement with the KEY= option to update a SAS data set. In the program below, when **_IORC_** has the value *_SOK*, the observation is updated. When **_IORC_** has the value *_DSENOM*, no matching observation is found, so the observation is appended to the data set by the OUTPUT statement and **_ERROR_** is reset to *0* in the do loop.

```
data master;
   set transaction;
   modify master key = id;
   if _IORC_=%sysrc(_sok) then
      do;
         a = b;
         replace;
      end;
   else
      if _IORC_=%sysrc(_dsenom) then
         do;
            output;
            _ERROR_ = 0;
         end;
run;
```

Tip For more information about the **_IORC_** automatic variable and *%SYSRC*, see information about error-checking tools in the SAS documentation.

Understanding Integrity Constraints

Overview

No that you know how to modify data in place, you might be wondering how you can protect or ensure the integrity of your data when it is modified. Integrity constraints are rules that you can specify in order to restrict the data values that can be stored for a variable in a data set. SAS enforces integrity constraints when values associated with a variable are added, updated, or deleted using techniques that modify data in place, such as

- a DATA step with the MODIFY statement
- an interactive data editing window
- PROC SQL with the INSERT INTO, SET, or UPDATE statements
- PROC APPEND.

When you add an integrity constraint to the table that contains data, SAS checks all data values to determine whether they satisfy the constraint before the constraint is added.

Type	Action

CHECK	ensures that a specific set or range of values are the only values in a column. It can also check the validity of a value in one column based on a value in another column within the same row.
NOT NULL	guarantees that a column has non-missing values in each row.
UNIQUE	enforces uniqueness for the values of a column.
PRIMARY KEY	uniquely defines a row within a table, which can be a single column or a set of columns. A table can have only one primary key. The PRIMARY KEY constraint includes the attributes of the NOT NULL and UNIQUE constraints.
FOREIGN KEY	links one or more rows in a table to a specific row in another table by matching a column or set of columns in one table with the PRIMARY KEY defined in another table. This parent/child relationship limits modifications made to both tables. The only acceptable values for a FOREIGN KEY are values of the PRIMARY KEY or missing values.

Note When you place integrity constraints on a SAS data set, you specify the type of constraint that you want to create. Each constraint has a different action.

You can use integrity constraints in two ways, *general* and *referential*. General constraints operate within a data set, and referential constraints operate between data sets.

General Integrity Constraints

General integrity constraints enable you to restrict the values of variables within a single data set. The following four integrity constraints can be used as general integrity constraints:

- CHECK
- NOT NULL
- UNIQUE
- PRIMARY KEY.

Note A PRIMARY KEY constraint is a general integrity constraint as long as it does not have any FOREIGN KEY constraints referencing it. When PRIMARY KEY is used as a general constraint it is simply a shortcut for assigning the NOT NULL and UNIQUE constraints.

Referential Integrity Constraints

Referential constraints enable you to link the data values of a column in one data set to the data values of columns in another data set. You create a referential integrity constraint when a FOREIGN KEY integrity constraint in one data set references a PRIMARY KEY integrity constraint in another data set. To create a referential integrity constraint, you must follow two steps:

1. Define a PRIMARY KEY constraint on the first data set.
2. Define a FOREIGN KEY constraint on other data sets.

Placing Integrity Constraints on a Data Set

Overview

Integrity constraints can be created using

- the DATASETS procedure
- the SQL procedure.

Although you can use either procedure to create integrity constraints on existing data sets, you must use PROC SQL if you want to create integrity constraints at the same time that you create the data set. In this chapter you learn to use PROC DATASETS to place integrity constraints on an existing data set.

General form, DATASETS procedure with the IC CREATE statement:

```
PROC DATASETS LIB=libref<NOLIST>;
     MODIFY SAS-data-set;
```

```
      IC CREATE constraint-name=constraint
         <MESSAGE='Error Messaged'>;
QUIT;
```

where

libref

is the library in which the data set is stored. If you do not specify the LIB= option, the procedure uses the Work library.

NOLIST

suppresses the directory listing.

SAS-data-set

is the name of the data set to which you want to apply the integrity constraint.

constraint-name

is any name that you want to give the integrity constraint.

constraint

is the type of constraint that you are creating, specified in the following format:

- NOT NULL *(variable)*
- UNIQUE *(variables)*
- CHECK *(where-expression)*
- PRIMARY KEY *(variables)*
- FOREIGN KEY *(variables)* REFERENCES *table-name*.

Error Message

is an optional message that you want the user to determine whether the constraint is violated.

Note You can use *IC* or *INTEGRITY CONSTRAINT* interchangeably.

Tip To learn how to create integrity constraints using the SQL procedure, see "Creating and Managing Tables Using PROC SQL" on page 175.

Example

Suppose you have a data set that contains route information and passenger capacity for each class in an airline. You need to create integrity constraints to ensure that when the data set is updated

- the route ID number is both unique and required (PRIMARY KEY)
- the capacity for business class passengers must either be missing or be greater than the capacity for first class passengers (CHECK).

In the code below, the IC CREATE statement is used to create two general integrity constraints on variables in the data set *Capinfo*:

- The PRIMARY KEY constraint is placed on the **`RouteID`** variable. This constraint ensures that when values of **`RouteID`** are updated, they must be unique and nonmissing.

  Note The same effect could be achieved by applying both the UNIQUE and NOT NULL constraints, but the PRIMARY KEY constraint is used as a shortcut.

- The CHECK constraint uses the WHERE expression to ensure that the only values of **`CapBusiness`** that are allowed are those greater than **`Cap1st`** or missing.

Note If you choose to run this example, you must copy the data set *Capinfo* from the *Sasuser* library to the *Work* library.

```
proc datasets nolist;
   modify capinfo;
   ic create PKIDInfo=primary key(routeid)
      message='You must supply a Route ID Number';
   ic create Class1=check(where=(cap1st<capbusiness or capbusiness=.))
      message='Cap1st must be less than CapBusiness';
quit;
```

Notice that the NOLIST option is used to prevent a listing of the Work directory that PROC DATASETS generally produces. When the constraint is created, a message is written to the SAS log.

Table 18.1: SAS Log

```
45 modify capinfo;
46 ic create PKIDInfo = primary key (routeid)
47 message = 'You must supply a Route ID Number';
NOTE: Integrity constraint PKIDInfo defined.
48 ic create Class1 = check (where = (cap1st < capbusiness
49 or capbusiness = .)) message = 'Cap1st must be less
50 than CapBusiness';
NOTE: Integrity constraint Class1 defined.
51 run;
```

Note For the UNIQUE and PRIMARY KEY constraints, SAS builds indexes on the columns involved if an appropriate index does not already exist. Any index created by an integrity constraint can be used for other purposes, such as WHERE processing or the KEY= option in a SET statement.

Tip For more information about creating integrity constraints, see the SAS documentation for the DATASETS procedure.

How Constraints Are Enforced

Once integrity constraints are in place, SAS enforces them whenever you modify values in the data set in place. Techniques for modifying data in place include using

- a DATA step with the MODIFY statement
- interactive data editing windows
- PROC SQL with the INSERT INTO, SET, or UPDATE statements
- PROC APPEND.

Example

The code in the previous example placed a check constraint on **`Cap1st`** and **`CapBusiness`** to ensure that values for the capacity in business class were either greater than first class or missing. Suppose that you ran the following program to triple the capacity in first class. This would probably violate the check constraint for some observations.

```
data capinfo;
   modify capinfo;
   cap1st=cap1st*3;
run;
```

The observations that failed to pass the integrity constraint are written to the SAS log. As you can see, all these observations would have had values of **`Cap1st`** greater than those of **`CapBusiness`**.

Table 18.2: SAS Log

```
FlightID=IA00100 RouteID=0000001 Origin=RDU Dest=LHR Cap1st=42
CapBusiness=30 CapEcon=163
```

```
_ERROR_=1 _IORC_=660130 _N_=1
FlightID=IA00201 RouteID=0000002 Origin=LHR Dest=RDU Cap1st=42
CapBusiness=30 CapEcon=163
_ERROR_=1 _IORC_=660130 _N_=2
FlightID=IA00300 RouteID=0000003 Origin=RDU Dest=FRA Cap1st=42
CapBusiness=30 CapEcon=163
_ERROR_=1 _IORC_=660130 _N_=3
FlightID=IA00400 RouteID=0000004 Origin=FRA Dest=RDU Cap1st=42
CapBusiness=30 CapEcon=163
_ERROR_=1 _IORC_=660130 _N_=4
FlightID=IA02900 RouteID=0000029 Origin=SFO Dest=HNL Cap1st=42
CapBusiness=30 CapEcon=163
_ERROR_=1 _IORC_=660130 _N_=29
FlightID=IA03000 RouteID=0000030 Origin=HNL Dest=SFO Cap1st=42
CapBusiness=30 CapEcon=163
_ERROR_=1 _IORC_=660130 _N_=30
FlightID=IA03300 RouteID=0000033 Origin=RDU Dest=ANC Cap1st=42
CapBusiness=30 CapEcon=163
_ERROR_=1 _IORC_=660130 _N_=33
FlightID=IA03400 RouteID=0000034 Origin=ANC Dest=RDU Cap1st=42
CapBusiness=30 CapEcon=163
_ERROR_=1 _IORC_=660130 _N_=34
NOTE: There were 50 observations read from the data set WORK.CAPINFO.
NOTE:  The data set WORK.CAPINFO has been updated. There were 42
      observations rewritten, 0 observations added and 0
      observations deleted.
NOTE: There were 8 rejected updates, 0 rejected adds, and 0 rejected
      deletes.
```

If you used the VIEWTABLE window or another window to make this update, SAS would have displayed the error message defined for the integrity constraint.

Note Rejected observations can be collected in a special file using the audit trail functionality that you will learn about later in this chapter.

Copying a Data Set and Preserving Integrity Constraints

The APPEND, COPY, CPORT, CIMPORT, and SORT procedures preserve integrity constraints when their operation results in a copy of the original data file. Integrity constraints are also preserved if you copy a data set using the SAS Explorer window.

Tip For more information about preserving integrity constraints, see the SAS documentation.

Documenting Integrity Constraints

Overview

To view the descriptor portion of your data, including the integrity constraints that you have placed on a data set, you can use the CONTENTS statement in the DATASETS procedure.

General form, DATASETS procedure with the CONTENTS statement:

```
PROC DATASETS LIB=libref<NOLIST>;
     CONTENTS DATA=SAS-data-set;
QUIT;
```

where

libref

is the library in which the data set is stored.

NOLIST

suppresses the directory listing.

SAS-data-set

is the name of the data set that you want information about.

Note The CONTENTS statement in the DATASETS procedure results in the same information as the CONTENTS procedure.

Example

The following code displays information about the *Capinfo* data set, including the integrity constraints that were added to this data set in the last example. Notice that the NOLIST option is used where to suppress the listing of all data sets in the Work library. With this option, only the information for the *Capinfo* data set is listed.

```
proc datasets nolist;
   contents data=capinfo;
quit;
```

Only the integrity constraints portion of the output is shown below.

Alphabetic List of Integrity Constraints

#	Integrity Constraint	Type	Variables	Where Clause	User Message
1	Class1	Check		(Cap1 s1<CapBusiness) or (CapBusiness=.)	First Class Capacity must be less than Business Capacity
2	PKIDInfo	Primary Key	RouteID		You must supply a Route ID Number

Removing Integrity Constraints

Overview

To remove an integrity constraint from a data set, use the DATASETS procedure with the IC DELETE statement.

General form, DATASETS procedure with the IC DELETE statement:

```
PROC DATASETS LIB=libref<NOLIST>;
      MODIFY SAS-data-set;
      IC DELETE constraint-name;
QUIT;
```

where

libref

is the name of the library in which the data set is stored. If you do not specify the LIB= option, the procedure uses the *Work* library.

NOLIST

suppresses the directory listing.

SAS-data-set

is the name of the data set that has the integrity constraint.

constraint-name

is the name of the integrity constraint that you want to delete.

Example

The code below removes the integrity constraints on the *Capinfo* data set:

```
proc datasets;
    modify capinfo;
    ic delete pkidinfo;
    ic delete class1;
 quit;
```

A message is written to the SAS log when the integrity constraint is deleted.

Table 18.3: SAS Log

```
53      modify capinfo;
54      ic delete pkidinfo;
NOTE: Integrity constraint PKIDInfo deleted.
55      ic delete class1;
NOTE: All integrity constraints defined on WORK.CAPINFO.DATA
      have been deleted.
56   run;NOTE: Integrity constraint PKIDInfo deleted.
```

Understanding Audit Trails

As you modify a data set, you might want to track the changes that you make by using an audit trail. An audit trail is an optional SAS tile that logs modifications to a SAS table. Audit trails are used to track changes that are made to the data set in place. Specifically, audit trails track changes made with

- the VIEWTABLE window
- the data grid
- the MODIFY statement in the DATA step
- the UPDATE, INSERT, or DELETE statement in PROC SQL.

For each addition, deletion, and update to the data, the audit trail automatically stores a copy of the variables in the observation that was updated, and information such as who made the modification, what was modified, and when the modification was made. It can also store additional information in user-defined variables.

The following PROC CONTENTS output lists the variables in an audit trail file for a data set that has two variables, **A** and **B**. You will learn more about these variables later in this chapter.

#	Variable	Type	Len	Pos	Format	Label
1	A	Num	8	0		
2	B	Num	8	8		
3	Who	Char	20	16		Name
4	Why	Char	20	36		Reason
5	_ATDATETIME_	Num	8	56	DATETIME19.	
10	_ATMESSAGE_	Char	8	114		
6	_ATOBSNO_	Num	8	64		
9	_ATOPCODE_	Char	2	112		
7	_ATRETURNCODE_	Num	8	72		
8	_ATUSERID_	Char	32	80		

Caution Any procedure or action that replaces the data set (such as the DATA step, CREATE TABLE in PROC SQL, or SORT without the OUT= option) will delete the audit trail. Audit trails should not be deleted with system tools such as Windows Explorer.

A SAS table can have only one audit trail file. The audit trail file

- is a read-only file
- is created by PROC DATASETS
- must be in the same library as the data file associated with it
- has the same name as the data set it is monitoring, but with a member type of AUDIT.

Next we will consider how you initiate an audit trail on a SAS data set.

Initiating and Reading Audit Trails

Overview

You initiate an audit trail using the DATASETS procedure with the *AUDIT* and *INITIATE* statements.

General form, DATASETS procedure to initiate an audit trail:

```
PROC DATASETS LIB=libref<NOLIST>;
     AUDIT SAS-data-set <SAS-password>;
     INITIATE;
QUIT;
```

where

libref

is the name of the library where the data set to be audited resides

NOLIST

suppresses the directory listing

SAS-data-set

is the name of the SAS data set that you want to audit

SAS-password

is the SAS data set password, if one exists

INITIATE

begins the audit trail on the data set specified in the AUDIT statement.

Example

The following code initiates an audit trail on the data set *Capinfo*.

Note If you choose to run this example, you must copy the data set *Capinfo* from the Sasuser library to the *Work* library.

```
proc datasets nolist;
   audit capinfo;
   initiate;
quit;
```

Table 18.4: SAS Log

```
60   audit capinfo;
61   initiate;
WARNING: The audited data file WORK.CAPINFO.DATA is not
         password protected.
         Apply an ALTER password to prevent accidental
         deletion or replacement of it and any associated
```

```
           audit files.
62   quit;
NOTE: The data set WORK.CAPINFO.AUDIT has 0 observations
      and 13 variables.
```

Note The audit trail file uses the SAS password that is assigned to the parent data set. Therefore, it is recommended that you alter the password for the parent data set. Use the ALTER= data set option to assign an alter-password to a SAS data set or to access a read-, write-, or alter- protected SAS data set. If another password is used or no password is used, then the audit file is still created, but is not protected.

Tip For more information about audit trails, see the SAS documentation for the DATASETS procedure.

Reading Audit Trail Files

When the audit trail is initiated, it has no observations until the first modification is made to the audited data set. When the audit trail file contains data, you can read it with any component of SAS that reads a data set. To refer to the audit trail file, use the TYPE= data set option.

General form, TYPE= data set option to specify an audit file:

```
(TYPE=AUDIT)
```

Examples

The following PROC CONTENTS code displays the contents of the audit trial file:

```
proc contents data=mylib.sales (type=audit);
run;
```

The following PROC PRINT code lists the data in the audit trail file for the data set *Capinfo*:

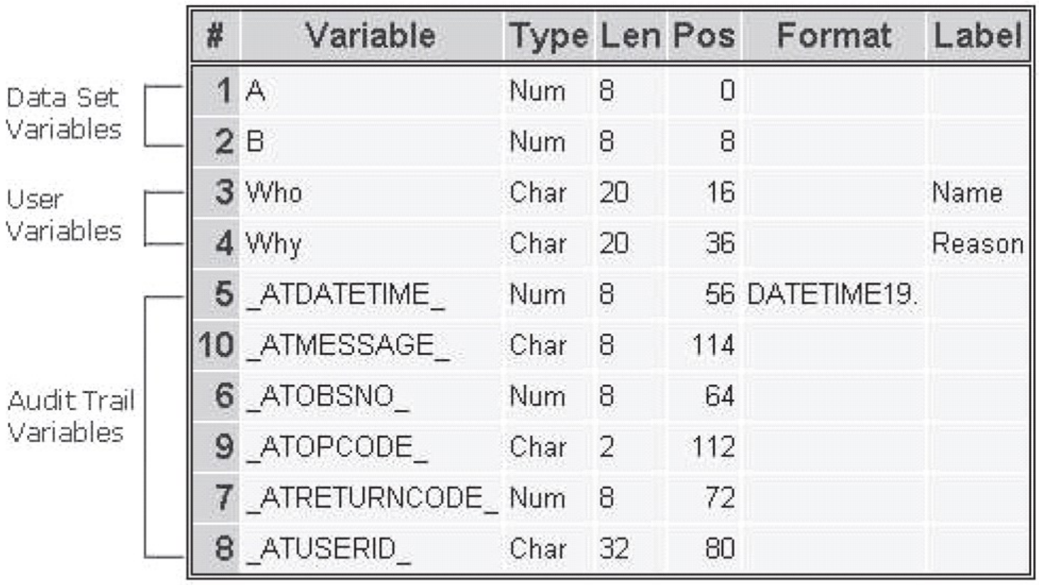
```
proc print data=capinfo (type=audit);
run;
```

Controlling Data in the Audit Trail

Overview

Now that you have seen how to initiate audit trails and read an audit trail file, consider the information the audit trail file contains. The audit trail file can contain three types of variables:

- *data set variables* that store copies of the columns in the audited SAS data set
- *audit trail variables* that automatically store information about data modifications
- *user variables* that store user-entered information.



#	Variable	Type	Len	Pos	Format	Label
1	A	Num	8	0		
2	B	Num	8	8		
3	Who	Char	20	16		Name
4	Why	Char	20	36		Reason
5	_ATDATETIME_	Num	8	56	DATETIME19.	
10	_ATMESSAGE_	Char	8	114		
6	_ATOBSNO_	Num	8	64		
9	_ATOPCODE_	Char	2	112		
7	_ATRETURNCODE_	Num	8	72		
8	_ATUSERID_	Char	32	80		

You can use additional statements in the PROC DATASETS step to control which variables appear in the audit trail. We will consider each of the three types of variables that can be found in an audit trail.

Data Set Variables

As you might expect, the audit trail file has the same set of variables that are in the audited data set. If the data set contains the variables **`A`** and **`B`**, the variables **`A`** and **`B`** are also in the audit trail file.

Next consider the audit trail variables that automatically store information about changes that you make to the data.

Audit Trail Variables

Audit trail variables automatically store information about data modifications. Audit trail variable names begin with AT followed by a specific string, such as DATETIME.

Audit trail variable	Information stored
`_ATDATETIME_`	date and time of a modification
`_ATUSERID_`	login user ID associated with a modification
`_ATOBSNO_`	observation number affected by the modification unless REUSE=YES
`_ATRETURNCODE_`	event return code
`_ATMESSAGE_`	SAS log message at the time of the modification
`_ATOPCODE_`	code describing the type of operation

Values of the _ATOPCODE_ Variable

The **`_ATOPCODE_`** variable contains a code that describes the type of operation that wrote the observation to the audit file. For example, if you modified all observations in an audited data set, the audit file would contain twice as many observations as the original data set. The audit file would contain one observation that matched the original observation with an **`_ATOPCODE_`** value of *DR*, and one updated observation with an **`_ATOPCODE_`** value of *DW*.

The table below shows the possible values of the **`_ATOPCODE_`** variable.

ATOPCODE	Event
DA	added data record image
DD	deleted data record image
DR	before-update record image

DW	after-update record image
EA	observation add failed
ED	observation delete failed
EU	observation update failed

You can define what information is stored in the audit file by using the LOG statement when you initiate the audit trail.

Using the LOG Statement to Control the Data in the Audit Trail

When you initiate an audit trail, options in the LOG statement determine the type of entries stored in the audit trail, along with their corresponding **_ATOPCODE_** values. The ERROR_IMAGE option controls E operation codes. The BEFORE_IMAGE option controls the DR operation code, and the DATA_IMAGE option controls all other D operation codes. If you omit the LOG statement when you initiate the audit trail, the default behavior is to log all images.

General form, LOG statement:

LOG <*audit-settings*>;

where

audit-settings

are any of the following:

- BEFORE_IMAGE=*YES*|*NO* controls storage of before-update record images (the 'DR' operation).
- DATA_IMAGE=*YES*|*NO* controls storage of after-update record images (for example, other operations starting with 'D').
- ERROR_IMAGE=*YES*|*NO* controls storage of unsuccessful update record images (for example, operations starting with 'E').

Example

The following code initiates an audit trail on the data set *Capinfo* but stores only error record images. This means that the audit file will contain only records where an error occurred. The **_ATOPCODE_** values can be only *EA, ED*, and *EU*.

Note If you choose to run this example, you must copy the data set *Capinfo* from the *Sasuser* library to the *Work* library.

```
proc datasets nolist;
   audit capinfo;
   initiate;
   log data_image=NO before_image=NO;
quit;
```

User Variables

User variables allow the person editing the file to enter information about changes they are making to the data. Although the data values are stored in the audit file, you can update them in the data set like any other variable.

User variables are created by using the USER_VAR statement in the audit trail specification.

General form, USER_VAR statement:

USER_VAR *variable-name* <$><*length*><LABEL='*variable-label*'>;

where

variable-name

is the name of the user variable you are creating.

$

indicates the variable is a character variable.

length

specifies the length of the variable (the default is 8).

variable-label

specifies a label for the variable enclosed in quotation marks.

Note You can create more than one user variable in a single USER_VAR statement.

User variables are unique in SAS in that they are stored in one file (the audit file) and opened for update in another file (the data set). When the data set is opened for update, the user variables display, and you can edit them as if they are part of the data set.

Example

Suppose you must monitor the updates for the data set *Capinfo*. The following code initiates an audit trail for the data set *Capinfo* and creates two user variables, **`who`** and **`why`**, to store who made changes to the data set and why the changes were made.

Note If you choose to run this example, you must copy the data set *Capinfo* from the *Sasuser* library to the *Work* library.

```
proc datasets nolist;
   audit capinfo;
   initiate;
   user_var who $20 label = 'Who made the change'
            why $20 label = 'Why the change was made';
quit;
```

Once these user variables are set up, they are retrieved from the audit trail and displayed when the data set is opened for update. You can enter data values for the user variables as you would for any data variable. The data values are saved to the audit trail as each observation is saved. The user variables are not available when the data is opened for browsing or printing. To rename a user variable or modify its attributes, you modify the data set, not the audit file.

Controlling the Audit Trail

Overview

Once you activate an audit trail, you can suspend and resume logging, and terminate (delete) the audit trail by resubmitting a PROC DATASETS step with additional statements. You use the DATASETS procedure to suspend and then resume the audit trail. You also use this procedure to delete or terminate an audit trail.

General form, DATASETS procedure to suspend, resume, or terminate an audit trail:

```
PROC DATASETS LIB=libref<NOLIST>;
     AUDIT SAS-data-set <SAS-password>;
     SUSPEND | RESUME | TERMINATE;
QUIT;
```

where

libref

is the name of the library where the table to be audited resides.

NOLIST

suppresses the directory listing.

SAS-data-set

is the name of the SAS data set that you want to audit.

SAS-password

is the SAS data file password, if one exists.

SUSPEND

suspends event logging to the audit file, but does not delete the audit file.

RESUME

resumes event logging to the audit file, if it was suspended.

TERMINATE

terminates event logging and deletes the audit file.

Tip Because each update to the data file is also written to the audit file, the audit trail can negatively impact system performance. You might want to consider suspending the audit trail for large, regularly scheduled batch updates.

Example

The following code terminates the audit trail on the data set *Capinfo*.

Note If you choose to run this example, you must copy the data set *Capinfo* from the *Sasuser* library to the *Work* library.

```
proc datasets nolist;
   audit capinfo;
   terminate;
quit;
```

A message is written to the log when the audit trail is terminated.

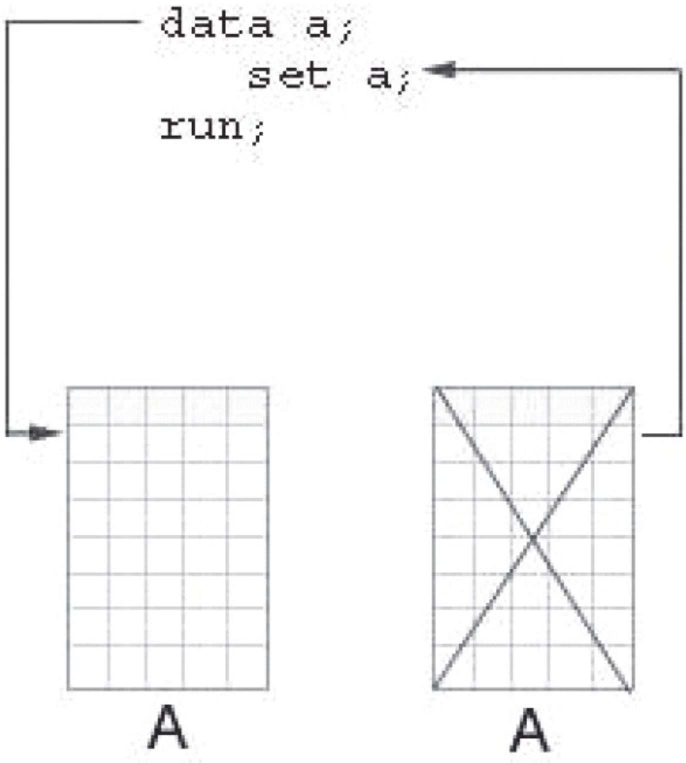
Table 18.5: SAS Log

```
65   audit capinfo;
66   terminate;
NOTE: Deleting WORK.CAPINFO (memtype=AUDIT).
67 quit;
```

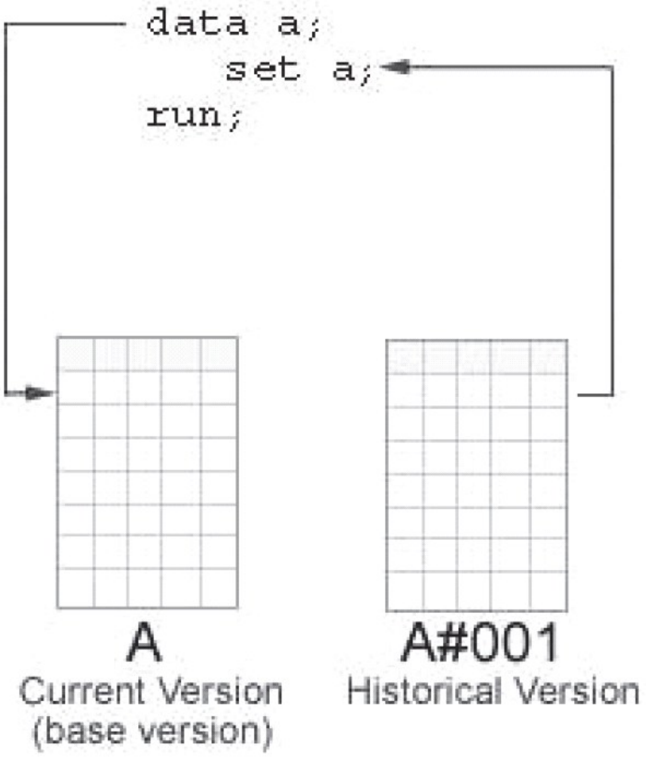
Understanding Generation Data Sets

You have learned that you can keep an audit trail to track observation updates made to an individual data set in place. However, if you replace the data set, the audit trail is lost. Generation data sets allow you to maintain multiple versions or generations of a SAS data set. A new generation is created each time the file is replaced.

By default, *generation data sets are not in effect*. As the SAS data set *A* is replaced, there are two copies of *A* in the SAS library. When the DATA step completes execution, SAS removes the original copy of the data set *A* from the library.



When *generation data sets are in effect* and the SAS data set *A* is replaced, there are two copies of *A* in the SAS library. When the DATA step completes execution, SAS keeps the original copy of the SAS data set *A* in the library and renames it.



Each generation of a generation data set is stored as part of a generation group. Each generation data set in a generation group has the same root member name, but each has a different version number. The most recent version is called the base version. When generations are in effect, SAS filenames are limited to 28 characters. The last four characters are reserved for the version numbers.

Note Generation data sets are not supported on VMS.

Initiating Generation Data Sets

Overview

To initiate generation data sets and to specify the maximum number of versions to maintain, you use the output data set option *GENMAX*= when creating or replacing a data set. If the data set already exists, you can use the GENMAX= option with the DATASETS procedure and the MODIFY statement.

General form, DATASETS procedure and MODIFY statement with the GENMAX= option:

```
PROC DATASETS LIB=libref<NOLIST>;
     MODIFY SAS-data-set (GENMAX=n);
QUIT;
```

where

libref

is the library that contains the data you want to modify.

NOLIST

suppresses the directory listing.

SAS-data-set

is the name of the SAS data set that you want to modify.

n

is the number of historical versions you want to keep, including the base version:

- *n=0*, no historical versions are kept (this is the default).
- *n>0*, the specified number of versions of the file will be kept. The number includes the base version.

Example

The following DATASETS procedure modifies the data set *Cargorev* and requests that up to four versions be kept (one base version and three historical versions).

Note If you choose to run this example, you must copy the data set *Cargorev* from the *Sasuser* library to the *Work* library.

```
proc datasets nolist;
   modify cargorev (genmax=4);
quit;
```

No message is written to the log when you specify the GENMAX= option.

Creating Generation Data Sets

Remember, new versions of a generation data set are created only when a data set is replaced, not when it is modified in place. To create new generations, use

- a DATA step with a SET statement
- a DATA step with a MERGE statement
- PROC SORT without the OUT= option
- PROC SQL with a CREATE TABLE statement.

Processing Generation Data Sets

Overview

Once you have a generation group that contains more than one generation data set, you might want to select a particular data set to process. To select a particular generation, you use the *GENNUM=* data set option.

General form, GENNUM= data set option:

GENNUM=*n*

where

n

specifies a particular historical version of a data set:

- *n>0* is an absolute reference to a historical version by its generation number.
- *n<0* is a relative reference to a historical version.
- *n=0* is the current version.

Examples

To print the current version of the data, you do not need to use the GENNUM= option. Simply use

```
proc print data=year;
run;
```

To print the youngest historical version, you can either specify the relative or absolute reference on the GENNUM= option, as shown:

```
proc print data=year(gennum=4);  /*absolute reference*/
run;
```

or

```
proc print data=year(gennum=-1);  /*relative reference*/
run;
```

You can also view information about a specific generation using the GENNUM= option with PROC CONTENTS, as shown:

```
proc contents data=year(gennum=-1);  /*relative reference*/
run;
```

Now that you have seen a few examples of using the GENNUM= option, consider how generation numbers change.

How Generation Numbers Change

When you use the GENNUM= option, you can refer to either the absolute or relative generation number. It's helpful to understand how generation numbers change so that you can identify the generation you want to process.

First, consider how SAS names generation data sets. The first time a data set with generations in effect is replaced, SAS keeps the replaced data set and appends a four-character version number to its member name, which includes the pound symbol (#) and a three-digit number. That is, for a data set named *A*, the replaced data set becomes *A#001*. When the data set is replaced for the second time, the replaced data set becomes *A#002* That is, *A#002* is the version that is chronologically closest to the base version. The table below shows the result after three replacements.

Data Set Name	Explanation
A	base (current) version
A#003	most recent (youngest) historical version

A#002	second most recent historical version
A#001	oldest historical version

The limit for version numbers that SAS can append is #999. After 1000 replacements, SAS rolls over the youngest version number to #000.

Now we will consider how absolute and relative generation numbers (specified on the GENNUM= option) change. Each time SAS creates a new generation, the absolute generation number increases sequentially. As older generations are deleted, their absolute generation numbers are retired.

In contrast, the relative generation number always refers to generations in relation to the base generation. The base or current generation is always 0 and -1 is the youngest historical version.

The following table shows data set names and their absolute and relative GENNUM= numbers.

Table 18.6: Data Set Names with GENNUM= Numbers

Interation	SAS Code	Data Set Names	GENNUM=Absolut e Reference	GENNUM=Relative Reference	Explanation
1	**data Year (genmax=3);**	*Year*	1	0	The data set *Year* is created, and three generations are requested.
2	**data Year;**	*Year* *Yera#001*	2 1	0 -1	*Year* us replaced. *Year* from iteration 1 is renamed *Year#001*.
3	**data Year;**	*Year* *Year#002* *Year#001*	3 2 1	0 -1 -2	*Year* is replaced. *Year* from iteration 2 is renamed *Year#002*.
4	**data Year;**	*Year* *Year#003* *Year#002*	4 3 2	0 -1 -2	*Year* is replaced. *Year* from iteration 3 is renamed *Year#003*. *Year#001* from iteration 1, which is the oldest, is deleted.
5	**data Year (genmax=2);**	*Year* *Year#004*	5 4	0 -1	*Year* is replaced, and the number of generations is changed to 2. *Year* from iteration 4 is renamed *Year#004*. The two oldest versions are deleted.

You have learned that you use PROC DATASETS to initiate generation data sets on an existing SAS data set. Once you have created generation data sets, you can use PROC DATASETS to perform management tasks such as

- deleting all or some of the generations
- renaming an entire generation group or any member of the group to a new base name.

General form, PROC DATASETS with the CHANGE and DELETE statements:

```
PROC DATASETS LIB=libref<NOLIST>;
     CHANGE SAS-data-set<(GENNUM=n)>=new-data-set-name;
     DELETE SAS-data-set<(GENNUM=n | HIST| ALL)>;
QUIT;
```

where

libref

is the library that contains the data you want to modify.

NOLIST

suppresses the directory listing.

SAS-data-set

is the name of the SAS data set you want to change or delete.

new-data-set-name

is the new name for the SAS data set in the *CHANGE* statement.

n

is the absolute or relative reference to a generation number.

HIST

refers to all generations except the base version.

ALL

refers to the base version and all generations.

Examples

The following code uses the CHANGE statement to rename the data set *SalesData* to *Sales*. If generations have been created, the base name of all generations will be renamed.

```
proc datasets library=quarter1 nolist;
   change salesData=sales;
quit;
```

The following code uses the GENNUM= option to rename only the second historical data set:

```
proc datasets library=quarter1 nolist;
   change sales(gennum=2)=newsales;
quit;
```

The following code deletes one historical version. This action might leave a hole in the generation group.

```
proc datasets library=quarter1 nolist;
   delete newsales(gennum=-1);
quit;
```

When you use the GENNUM= option with the DELETE statement, you can use the HIST and ALL keywords. The following code uses the *HIST* keyword to delete all of the historical versions:

```
proc datasets library=quarter1 nolist;
   delete newsales(gennum=HISTm);
quit;
```

The following code uses the *ALL* keyword in the GENNUM= option to delete all of the SAS data sets in a generation group:

```
proc datasets library=quarter1 nolist;
   delete newsales(gennum=ALL);
quit;
```

Tip For more information about using the DATASETS procedure to process data, see the SAS documentation.

Summary

Contents

This section contains the following topics.

- "Sample Programs" on page 693
- "Points to Remember" on page 694

Text Summary

Using the MODIFY Statement

When you use the MODIFY statement to modify a SAS data set, SAS does not create a second copy of the data as it does when you use the SET, MERGE, or UPDATE statements. The descriptor portion of the SAS data set stays the same and the updated observation is written to the data set in the location of the original observation.

Modifying All Observations in a SAS Data Set

You can use the MODIFY statement with an assignment statement to modify all the observations for a variable in a SAS data set.

Modifying Observations Using a Transaction Data Set

To modify a master data set using a transaction data set, you use the MODIFY statement with a BY statement to specify the variable you want to use to match. When you use the MODIFY/BY statements, SAS uses a dynamic WHERE clause to locate observations in the master data set. You can specify how missing values in the transaction data set are handled by using the UPDATEMODE= option in the MODIFY statement.

Modifying Observations Located by an Index

You can use the MODIFY statement with the KEY= option to name a simple or composite index for the SAS data set that is being modified. The KEY= argument retrieves observations from the SAS data file based on index values that are supplied by like-named variables in a transaction data set. If you have contiguous duplications in the transaction data set such that there is no match in the master data set, you can use the UNIQUE option to cause a KEY= search to always begin at the top of the index file for each duplicate transaction.

Controlling the Update Process

The way SAS writes observations to a SAS data set when the DATA step contains a MODIFY statement depends on whether certain other statements are present. If no statement is present, SAS writes the current observation to its original place in the SAS data set. This occurs as the last action in the step as though a REPLACE statement were the last statement in the step. However, you can override this default behavior by explicitly adding the OUTPUT, REPLACE, or REMOVE statement.

You can use the automatic variable **`_IORC_`** with the %SYSRC autocall macro to test for specific I/O error conditions that are created when you use the BY statement or the KEY= option in the MODIFY statement. The automatic variable **`_IORC_`** contains a return code for each I/O operation that the MODIFY statement attempts to perform. The best way to test for values of **`_IORC_`** is with the mnemonic codes that are provided by the *SYSRC* autocall macro.

Placing Integrity Constraints on a Data Set

Integrity constraints are rules that you can specify in order to restrict the data values that can be stored for a variable in a SAS data file. SAS enforces integrity constraints when values associated with a variable are added, updated, or deleted. You can place integrity constraints on an existing data set using the IC CREATE statement in the DATASETS procedure.

Documenting and Removing Integrity Constraints

You can view information about the integrity constraints on a data set using the CONTENTS statement in the DATASETS procedure. If you want to remove integrity constraints from a tile, you use the IC DELETE statement.

Initiating and Terminating Audit Trails

An audit trail is an optional SAS tile that logs modifications to a SAS table. You initiate an audit trail using the DATASETS procedure with the AUDIT and INITIATE statements. You also suspend, resume, and terminate audit trails using the DATASETS procedure. Once there is data in the audit trail file, you can read it with the TYPE= data set option.

Controlling Data in the Audit Trail

The audit trail file can contain three types of variables:

- data set variables that store copies of the columns in the audited SAS data file
- audit trail variables that automatically store information about data modifications
- user variables that store user-entered information.

You can use the LOG statement to control which types of records are written to an audit trail file.

Initiating Generation Data Sets

Each generation of a generation data set is stored as part of a generation group. A new generation is created each time the file is replaced. Each generation in a generation group has the same root member name, but each has a different version number. You initiate generation data sets by using the GENMAX= option to specify the number of generation data sets to keep.

Processing Generation Data Sets

To select a particular generation to process, you use the GENNUM= option. GENNUM= is an input/update data set option that identifies which generation to open. The GENNUM can be a relative or absolute reference to a generation within a generation group. You can rename or delete generations using the CHANGE and DELETE statements in a PROC DATASETS step.

Syntax

Modifying All Observations in a SAS Data Set

```
DATA SAS-data-set;
     MODIFY SAS-data-set;
     existing-variable - expression;
RUN;
```

Modifying a Master Data Set Using the BY Statement

```
MODIFY master-data-set transaction-data-set
    <UPDATEMODE=MISSINGCHECK | NOMISSINGCHECK>;
    BY key-variable;
```

Modifying a Master Data Set Using a Transaction Data Set and an Index

```
MODIFY master-data-set KEY=index </UNIQUE>;
```

Controlling the Update Process

```
OUTPUT;
REPLACE;
REMOVE;
```

Using PROC DATASETS to Create Integrity Constraints, Generation Data Sets, and Audit Trails

```
PROC DATASETS <LIB=libref> <NOLIST> ;
     IC CREATE <constraint-name=>constraint-type
        <MESSAGE= 'Error Message'>;
     IC DELETE constraint-name;
     MODIFY SAS-data-set (GENMAX=n);
     AUDIT SAS-data-file <password>;
          INITIATE;
          <LOG<audit-settings>>;
          <USER_VARvariable-name<$><length><LABEL='variable-label'>>;
          SUSPEND | RESUME | TERMINATE;
     CONTENTS data=SAS-data-set;
QUIT;
```

Using _IORC_ with %SYSRC

```
IF _IORC_=%SYSRC(mnemonic) THEN…
```

Specifying an Audit Trail File

```
(TYPE=AUDIT)
```

Using PROC DATASETS to Rename orDelete Generation Data Sets

```
PROC DATASETS LIB=SAS-library <NOLIST>;
```

```
     CHANGE SAS-data-set<(GENNUM=n)>=new-data-set-name;
     DELETE SAS-data-set<(GENNUM=n| HIST | ALL)>;
QUIT;
```

Sample Programs

Modifying a Data Set Using the MODIFY Statement with a BY Statement and with the KEY= Option

```
data capacity;
   modify capacity sasuser.newrtnum;
   by flightid;
run;

data cargo99;
   set sasuser.newcgnum (rename =
       (capcargo = newCapCargo
       cargowgt = newCargoWgt
       cargorev = newCargoRev));
   modify cargo99 key=flghtdte;
   capcargo = newcapcargo;
   cargowgt = newcargowgt;
   cargorev = newcargorev;
run;
```

Placing Integrity Constraints on Data

```
proc datasets nolist;
   modify capinfo;
   ic create PKIDInfo=primary key(routeid)
      message='You must supply a Route ID Number';
   ic create Class1=check(where=(cap1st<capbusiness
                                 or capbusiness=.))
      message='Cap1st must be less than CapBusiness';
quit;
```

Initiating an Audit Trail

```
proc datasets nolist;
   audit capinfo;
   initiate;
quit;
```

Initiating Generation Data Sets

```
proc datasets nolist;
   modify cargorev (genmax=4);
quit;
```

Points to Remember

- The MODIFY statement in a DATA step is used to make updates to a SAS data set in place. The descriptor portion of the SAS data set cannot be changed.
- Integrity constraints are enforced only when modifications are made to the data. If the data set is replaced, integrity constraints are lost.
- Audit trail files track changes made to data sets *in place* with
  - the MODIFY statement in the DATA step
  - the UPDATE, INSERT, or DELETE statement in PROC SQL.
- Generation data sets are used to track changes that are made when a data set is replaced by
  - using the SET, MERGE, or UPDATE statements in the DATA step
  - sorting data in place with PROC SORT
  - using the CREATE TABLE statement in PROC SQL.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. Which type of integrity constraint would you place on the variable **`StoreID`** to ensure that there are no missing values and that there are no duplicate values? ?

a. UNIQUE

b. CHECK

c. PRIMARYKEY

d. NOT NULL

2. Which code creates an audit trail on the SAS data set *Reports.Quarter1*? ?

a.
```
a. proc datasets nolist;
     audit quarter1;
     initiate;
  quit;
```

b.
```
b. proc datasets lib=reports nolist;
     audit initiate reports.quarter1;
  quit;
```

c.
```
c. proc datasets lib=reports nolist;
     initiate audit quarter1;
  quit;
```

d.
```
d. proc datasets lib=reports nolist;
     audit quarter1;
     initiate;
  quit;
```

3. Which DATA step uses the transaction data set *Records.Overnight* to update the master data set *Records.Snowfall* by **`accumAmt`**? ?

a.
```
a. data records.snowfall;
     modify records.snowfall records.overnight
     key=accumAmt;
   run;
```

b.
```
b. data records.snowfall;
     modify records.overnight records.snowfall;
     by accumAmt;
  run;
```

c.
```
c. data records.snowfall;
     modify records.snowfall records.overnight;
     by accumAmt;
  run;
```

d.
```
d. data records.snowfall;
     modify records.snowfall records.overnight;
     update accumAmt;
  run;
```

4. The automatic variable **`_IORC_`** is created when you use the MODIFY statement with a BY statement or the KEY= option. How can you use the value of **`_IORC_`**? ?

a. to determine whether the index specified on the KEY= option is a valid index

b. to determine the number of observations that were updated in the master data set

c. to determine the status of the I/O operation

d. to determine the number of observations that could *not* be updated in the master data set

5. Which PROC DATASETS step creates an integrity constraint named val_age on the data set *Survey* to ensure that values of the variable **`age`** are greater than or equal to 18? ?

a.
```
a. proc datasets nolist;
     modify age;
     ic create val_age=check(where=(age>=18));
  quit;
```

b.
```
b. proc datasets nolist;
     modify Survey;
     ic create val_age=check(age>=18);
  quit;
```

c.
```
c. proc datasets nolist;
     modify survey;
     integrity constraint
               val_age=check(where=(age>=18));
  quit;
```

d.
```
d. proc datasets nolist;
     modify survey;
     ic create val_age=check(where=(age>=18));
  quit;
```

6. Which statement about using the MODIFY statement in a DATA step is true? ?

a. MODIFY creates a second copy of the data while variables in the data are being matched with a WHERE clause and then deletes the second copy.

b. You cannot modify the descriptor portion of the data set using the MODIFY statement.

c. You can use the MODIFY statement to change the name of a variable.

d. If the system terminates abnormally while a DATA step that is using the WHERE statement is processing, SAS automatically saves a copy of the unaltered data set.

7. Which of the following statements about audit trails is true? ?

a. They create historical versions of data so that a copy of the data set is saved each time the data is replaced.

b. They record information about changes to observations in a data set each time the data set is replaced.

c. They record information about changes to observations in a data set each time the data is modified in place.

d. The audit trail file has the same name as the SAS data file it is monitoring, but has #AUDIT at the end of the data set name.

8. Which code initiates generation data sets on the existing SAS data set *Sasuser.Amounts* and specifies that five historical versions are saved in addition to the base version? ?

a.
```
a. proc datasets lib=sasuser nolist;
     modify Amounts (genmax=6);
  quit;
```

b.
```
b. proc datasets lib=sasuser nolist;
     modify Amounts (genmax=5);
  quit;
```

c.
```
c. proc datasets lib=sasuser nolist;
     modify Amounts (gennum=6);
  quit;
```

d.
```
d. proc datasets lib=sasuser nolist;
     modify Amounts (gennum=5);
  quit;
```

9. Which statement about using the KEY= option in the MODIFY statement is true? ?

a. SAS locates the variables to update using the index specified in the KEY= option and then automatically overlays nonmissing transaction values as it does when you use the MODIFY/BY statements.

b. When you use the KEY= option, you must explicitly state the update that you want to make. SAS does not automatically overlay nonmissing transaction values.

c. The KEY= option is used to specify a variable to match for updating observations.

d. The index named in the KEY= option must be a simple index.

10. Which code deletes all generations of the data set *Sasuser.Amounts* including the base data set? ?

a.
```
a. proc datasets lib=sasuser nolist;
     delete amounts (gennum=ALL);
  quit;
```

b.
```
b. proc datasets lib=sasuser nolist;
     delete amounts (gennum=HIST);
  quit;
```

c.
```
c. proc datasets lib=sasuser nolist;
     delete amounts (gennum=0);
  quit;
```

d.
```
d. proc datasets lib=sasuser nolist;
     delete amounts;
  quit;
```

Answers

1. Correct answer: c

The PRIMARY KEY integrity constraint includes both the NOT NULL and UNIQUE constraints.

2. Correct answer: d

To initiate an audit on an existing SAS data set with the DATASETS procedure, you specify the data set in the AUDIT statement, and then you specify the INITIATE statement. You specify the library with the LIB= option.

3. Correct answer: c

In the MODIFY statement, you specify the master data set followed by the transaction data set. Then you specify the variable in the BY statement.

4. Correct answer: c

The value of _IORC_ is a numeric return code that indicates the status of the most recently executed I/O operation. Checking the value of this variable allows you to detect abnormal I/O conditions and direct execution in particular ways.

5. Correct answer: d

   In the MODIFY statement, you list the SAS data set that you want to modify. Then you use the IC CREATE statement to create the integrity constraint. This integrity constraint is a CHECK constraint and you use a WHERE clause to specify the condition that the variable values must meet.

6. Correct answer: b

   The MODIFY statement in a DATA step can be used only to modify the values in a data set. It cannot be used to modify the descriptor portion of the data set.

7. Correct answer: c

   Audit trails are used to track changes that are made to a data set in place.

8. Correct answer: a

   You use the DATASETS procedure and the MODIFY statement to specify a number of generation data sets for a data set. The GENMAX= option is used to specify the number of versions to save. The number you specify includes the base version.

9. Correct answer: b

   When you use the KEY= option, you must specify the update that you want to make to the data set.

10. Correct answer: a

    The keyword ALL is used to indicate that you want to delete all generations of the specified data set including the base version. The keyword HIST deletes the generation data sets, but saves the base version.